FINE-GRAIN PARALLELIZATION VERSUS THE WAVEFRONT METHOD

Alexander Aiken

Alexandru Nicolau

Department of Information and Computer Science

University of California, Irvine

Irvine, California 92717

Technical Report No.89-30

.2 4 1 - 0 4

Fine-Grain Parallelization Versus the Wavefront Method*

Alexander Aiken
IBM Almaden Research Center
San Jose, CA 95120

Alexandru Nicolau
Dept. of Information and Computer Science
UC Irvine
Irvine, CA 92717

Abstract

We develop a technique for extracting parallelism from ordinary (sequential) programs. The technique combines two fine-grain, instruction level code transformations to achieve effects similar to the wavefront method. Such effects were previously available only as transformations at coarser levels of granularity. By integrating the strengths of the wavefront method with the ability to extract fine-grain, irregular parallelism at the instruction level, our technique exploits previously untapped parallelism.

1 Introduction

Progress in parallelizing compiler technology is perhaps the most important factor in translating the promise of parallel computing—dramatically faster computation—into reality for most users. Much progress has been made, and many techniques developed for extracting parallelism from ordinary programs [Kuc76,AK82,FERN84,PW86,Cyd87] have been integrated into production tools.

Compile-time parallelization techniques fall roughly into two, largely disjoint classes. The first class, fine-grain, low-level parallelization, extracts irregular parallelism at the instruction level, but cannot deal well with parallelism at the level of loops[1]. The second class sacrifices irregular parallelism in favor of high-level, regular parallelism achieved by overlapping (partially or completely) loop iterations or full loops. Percolation Scheduling [Nic85] is an example of the first class of techniques; doacross [Cyt86] is an example of the second class. The strengths of these two approaches are complimentary; with the emergence of machines that can exploit both instruction-level and coarser forms of parallelism—such

*This work was supported in part by NSF grant CCR-87-04367, ONR grant N00014-86-K-0215, and the Cornell NSF Supercomputing Center.

[1] While simple loop-unwinding may alleviate this problem, it does not eliminate it.

as Multiflow's Trace, Cydrome's Cydra, Chopp, Alliant, Burton Smith's Horizon, and machines based on Intel's i860 chip—the integration of these levels of parallelism becomes important.

We show how to uniformly integrate fine-grain and coarse-grain parallelization of nested loops. We make use of two techniques: perfect pipelining [AN88b,Aik88], a transformation that bridges the gap between instruction level and iteration level parallelization for innermost loops, and loop quantization [Nic87,AN87], a transformation that exposes instruction-level parallelism across nested loops. The algorithm shares many of the properties of the wavefront method [Wol87], one of the most powerful high-level (nested-loop) parallelizing transformations, while also exploiting any irregular parallelism available at the fine-grain (instruction) level. The development illustrates the surprising expressive power of the fine-grain transformations and their relation to the wavefront method.

2 Basic Definitions

A set of nested loops L consists of loops labeled L_1 (outermost) to L_n (innermost). Each loop L_j has an associated index variable I_j. The *iteration space* of L consists of *iteration vectors* $\langle i_1, \ldots, i_n \rangle$ where each i_j is an assignment to I_j [Kuh80]. Iteration vectors are ordered lexicographically; this corresponds to the normal sequential execution order of iterations. We assume that $\langle 0, \ldots, 0 \rangle$ is the first iteration of any loop.

Programs are represented as *program-graphs* (also called control-flow graphs) where nodes contain zero or more statements. A statement is either an *assignment* or a *test* (an If-statement). Execution of a program begins at a distinguished start node and proceeds sequentially from node to node. A node is executed by evaluating the node's statements in parallel; the assignments update the store and the tests return the next node to be executed. The precise definition of the concurrent execution of tests is beyond the scope of this paper; details may be found in [Nic85,Aik88].

Program parallelization requires *dependency analysis*. Two program statements are dependent if one accesses a storage location that the other writes. Dependent statements may not be executed in parallel. Dependency information is usually represented by a *dependency graph*, where an edge between two statements represents a potential dependency

[KKP*81]. We say that two iterations of a loop are dependent if any pair of statements from the two iterations is dependent.

Our algorithm uses three low-level transformations of a program-graph: *unroll*, *move*, and *delete*. *Unroll* adds (unrolls) one copy of the original loop body at the end of the current loop body. We distinguish between the original and current loops because we alternately unroll and perform code motions. The *move* transformation moves a statement x from a node i to a predecessor of i if dependencies are not violated. The *delete* transformation deletes an empty node (a node with no statements) from the program-graph. A program is parallelized by application of these transformations, packing multiple statements into nodes for parallel execution. These transformations are based on the primitives of percolation scheduling. Complete descriptions of the transformations and the model of computation are in [AN88a].

For simplicity, we restrict the development to two nested loops with a single assignment in the loop body. The results apply directly to loops with any number of statements and arbitrary flow-of-control. Because of the simple flow-of-control in the example loops, we can adopt a representation more readable than program graphs. Statements are written in a standard high-level syntax. All statements on each line of a program are executed in parallel; successive lines are executed sequentially.

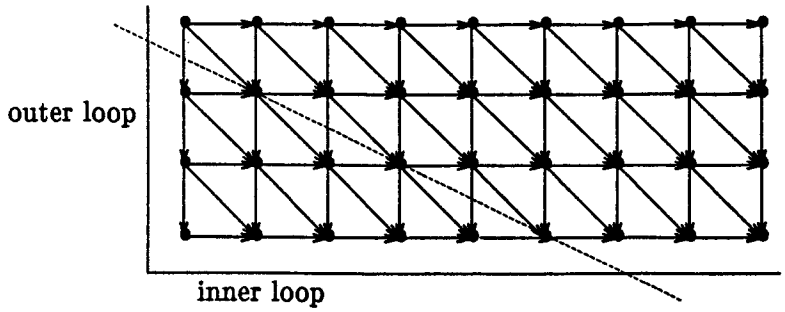
3 The Wavefront Method

The wavefront method [Mur71,Lam74,Kuh80,Wol82,Wol87] extracts parallelism from multiple nested loops in many cases where parallelism cannot be found in any single loop. The traditional derivation of the wavefront method involves finding a legal wavefront (i.e., a line in two dimensions, a plane or hyperplane in higher dimensions) through the iteration space. All iterations on the same wavefront may be executed in parallel. A legal wavefront preserves dependencies—two dependent iterations are executed in the same order as in the original loop. When a loop is executed in the wavefront ordering, iteration $\langle i, j \rangle$ is executed at wavefront (time) step $i * b + j$, where the wavefront angle is $\arctan(-1/b)$.[2]

[2]We depict the iteration space in the first quadrant of the $\langle i, j \rangle$ plane; other authors have chosen the fourth quadrant [Wol87].

```
for i ← 0 to N_j do
    for j ← 0 to N_i do
        A[i,j] ← A[i − 1, j + 1] + A[i − 1, j] + A[i, j + 1];
```

(a) A sample loop.



(b) The iteration space and optimal wavefront.

Figure 1: An example of the wavefront method.

The goal of the wavefront method is to find a wavefront that maximizes parallelism. Consider the loop in Figure 1a. Figure 1b shows the iteration space and dependencies; the optimal wavefront angle is 30°. We use a slightly different formulation of the wavefront to make comparison with our technique direct. The wavefront is expressed as a line with slope $-1/b$ for some positive integer b. (As in [Wol87], we do not consider the cases where the wavefront angle is 0° or 90°. In these cases, the wavefront method does not apply.) There are many potential wavefronts that cannot be expressed by these ratios; however, the optimal wavefront is always of this form. In Figure 1b, the optimal wavefront slope is $-1/2$.

The wavefront must preserve dependencies; a wavefront that is too steep may reverse the order of dependent iterations. The following definition describes all legal wavefronts.

Definition 3.1 (Legal Wavefronts) Let L be two nested loops, let $-1/b$ be a wavefront slope, and let $\langle i, j \rangle$ and $\langle i', j' \rangle$ be the iteration vectors of any dependent iterations. Then the wavefront is legal if:

$$\langle i, j \rangle < \langle i', j' \rangle \Leftrightarrow i * b + j < i' * b + j'$$

4 Loop Quantization

Loop Quantization [Nic87] is a technique for unrolling multiple nested loops. Loop unrolling provides a large number of instructions—the unrolled loop body—for scheduling by instruction-level transformations (such as trace scheduling [Fis79] or percolation scheduling [Nic85,AN88a]). Unrolling nested loops is important because parallelism may be present in outer loops and not in the inner loop, or even across several of the nested loops.

In general, loop quantization computes integers $k_1, \ldots, k_n$, where n is the number of nested loops. The original loop is unrolled k_1 times on the innermost loop, then this new loop body is unrolled k_2 times on the next innermost loop, and so on. The resulting loop $\hat{L}$ has an n-dimensional "box" of iterations of L as its loop body. All iterations in the box are executed before the box is shifted (by a "quantum" jump) along any of the dimensions. An example of a two by two quantization of the loop in Figure 1 is given in Figures 2a and 2b.[3]

Quantization alters the execution order of iterations of L and may therefore violate data dependencies. Conditions under which a quantization is legal are given in [AN87,Aik88]. The two by two quantization given in Figure 2b is illegal. As shown in Figure 3, the quantization box "cuts" a dependency illegally—iteration $\langle i, j \rangle$ must execute before iteration $\langle i+1, j-1 \rangle$.

Under certain conditions a *mitred quantization* can permit quantization where a normal quantization is illegal [AN87,Aik88]. Mitred quantization permits rhomboid quantization boxes instead of simple rectangular quantization boxes. For example, in Figure 4, the two by two mitred quantization is legal, because all dependent iterations are either included inside a quantized iteration or are satisfied by normal loop order execution in the quantized iteration space. Figure 2c shows the example loop after a two by two mitred quantization.

Mitred quantization computes the slant of the quantization box from the dependencies of the loops. For instance, in Figure 1, iteration $\langle i, j \rangle$ is dependent on iteration $\langle i+1, j-1 \rangle$. The *slope* of this dependency is $-1/1$. Mitred quantization selects as the slope of the sides of the quantization box the smallest negative slope of all dependencies. This produces a non-

[3] Quantized loops require a small amount of extra code to handle boundary conditions; for details see [AN87,Aik88].

```
for i ← 0 to N_i do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j] ← A[i − 1,j + 1] + A[i − 1,j] + A[i,j + 1];
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
        end
```

(a) Loops after inner loop is unrolled twice.

```
for i ← 0 to N_i by 2 do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j] ← A[i − 1,j + 1] + A[i − 1,j] + A[i,j + 1];
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
            A[i + 1,j] ← A[i,j + 1] + A[i,j] + A[i + 1,j + 1];
            A[i + 1,j + 1] ← A[i,j + 2] + A[i,j + 1] + A[i + 1,j + 2];
        end
```

(b) Loops after 2×2 quantization.

```
for i ← 0 to N_i by 2 do
    for j ← 0 to N_j by 2 do
        begin
            A[i,j + 1] ← A[i − 1,j + 2] + A[i − 1,j + 1] + A[i,j + 2];
            A[i,j + 2] ← A[i − 1,j + 3] + A[i − 1,j + 2] + A[i,j + 3];
            A[i + 1,j] ← A[i,j + 1] + A[i,j] + A[i + 1,j + 1];
            A[i + 1,j + 1] ← A[i,j + 2] + A[i,j + 1] + A[i + 1,j + 2];
        end
```

(c) Loops after 2×2 mitred quantization.

Figure 2: How quantization works.

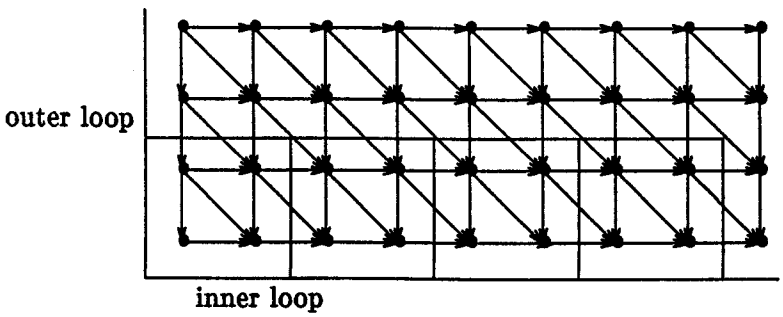


Figure 3: The two by two quantization is illegal.

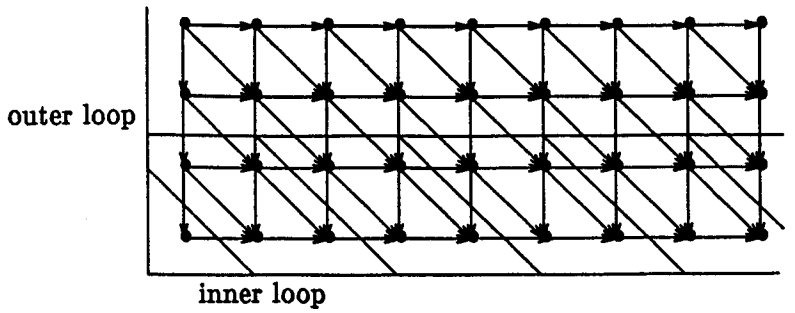


Figure 4: A legal mitred quantization of the loop in Figure 1.

trivial, legal quantization for loops encountered in practice. There is a strong relationship between mitred quantization and the wavefront method. Indeed, similar effects to mitred quantizations can be achieved by combining the unroll-and-jam of [CCK87] with the skewing produced by [Wol87].

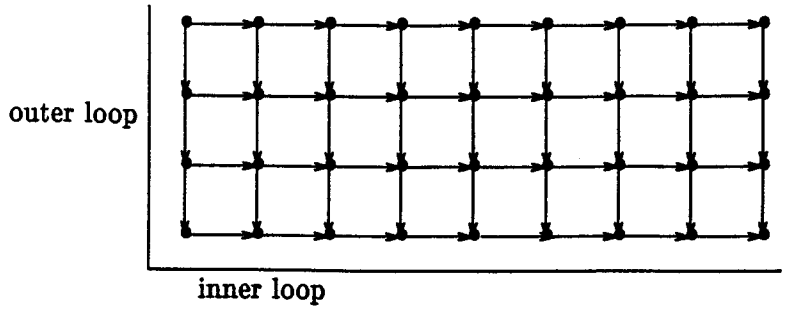
Lemma 4.1 Let L be two nested loops. Then the following statements about L are equivalent:

1. The optimal wavefront slope is $-1/b$.
2. The smallest negative slope of any dependency is $-1/(b-1)$.
3. A mitred quantization box has slope $-1/(b-1)$.

Proof: Let $-1/(b-1)$ be the slope of the mitred quantization. Trivially, this is also the smallest negative slope of any dependency. It is easy to show that the wavefront slope $1/b$ is legal by Constraint 3.1, because it preserves the dependency with the smallest negative slope. Furthermore, this is the best wavefront slope—a slope of $1/(b-1)$ would put two dependent iterations on the same wavefront. Finally, if $1/b$ is the best wavefront slope, then there must be a dependency of slope $-1/(b-1)$ and no dependency of smaller negative slope. If the optimal wavefront slope is $-1/1$, then normal (rectangular) quantization applies. □

```
for i ← 0 to N_i
    for j ← 0 to N_j
        A[i,j] ← (A[i+1,j] + A[i,j+1] + A[i-1,j] + A[i,j-1])/4;
```

(a) Gauss-Seidel iteration.



(b) The iteration space.

Figure 5: Another loop.

5 Perfect Pipelining

Quantization alone is insufficient to express the wavefront method using fine-grain parallelization. In principle, there is a fundamental difference. The wavefront method expresses unbounded parallelism—the number of iterations executable in parallel in the wavefront method is unrestricted, and parallelism grows with the size of the iteration space. In our code compaction model, each node in the program graph can contain only a finite number of statements. However, this is not a meaningful difference; programs must be executed on a machine with fixed resources—e.g., processors. We make the assumption that there is a k such that no more than k iterations can be executed in parallel. We derive an algorithm that exposes at least as much parallelism as the wavefront method for any machine size k.

There is a more serious barrier to achieving the effect of the wavefront method using fine-grain parallelization. Consider the standard Gauss-Seidel iteration loop in Figure 5; the iteration space with dependencies is shown in Figure 6. Assume that the target machine has sufficient resources to execute three iterations of this loop in parallel. The loop unrolled with a three by four quantization is given in Figure 7.[4] (We show below how to determine the amount of such unrollings.) The loop after compaction—every statement moved as far

[4]Normal quantization is adequate for this example.

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 4 do
        begin
            A[i,j] ← (A[i+1,j] + A[i,j+1]+
                A[i-1,j] + A[i,j-1])/4;
            A[i,j+1] ← (A[i+1,j+1] + A[i,j+2]+
                A[i-1,j+1] + A[i,j])/4;
            ⋮
            A[i,j+3] ← (A[i+1,j+3] + A[i,j+4]+
                A[i-1,j+3] + A[i,j+2])/4;
            ⋮
            A[i+2,j+2] ← (A[i+3,j+2] + A[i+2,j+3]+
                A[i+1,j+2] + A[i+2,j+1])/4;
            A[i+2,j+3] ← (A[i+3,j+3] + A[i+2,j+4]+
                A[i+1,j+3] + A[i+2,j+2])/4
        end;
```

Figure 6: Loop with three by four quantization.

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 4 do
        begin
            [i,j]
            [i,j+1]     [i+1,j]
            [i,j+2]     [i+1,j+1]   [i+2,j]
            [i,j+3]     [i+1,j+2]   [i+2,j+1]
            [i+1,j+3]   [i+2,j+2]
            [i+2,j+3]
        end;
```

Figure 7: Loop after quantization and compaction.

```
for i ← 0 to N_i by 3 do
    for j ← 0 to N_j by 6 do
        begin
            [i,j]
            [i,j+1]     [i+1,j]
            [i,j+2]     [i+1,j+1]   [i+2,j]
            [i,j+3]     [i+1,j+2]   [i+2,j+1]
            [i,j+4]     [i+1,j+3]   [i+2,j+2]
            [i,j+5]     [i+1,j+4]   [i+2,j+3]
            [i+1,j+5]   [i+2,j+4]
            [i+2,j+5]
        end;
```

Figure 8: Loop after further unrolling and compaction.

"up" as possible using move and delete—is given in Figure 8. For brevity, only the values of the index variables are given for each statement. Although full resource utilization is achieved in the middle of the loop body, utilization at the beginning and the end of the loop body is lower. Furthermore, regardless of the unrolling on either loop, after compaction there is always some start-up and wind-down code in the loop body.

The wavefront method also has start-up and wind-down periods where resources are less than fully utilized. However, for the wavefront method this occurs only once at the beginning and once at the end of the execution of the nested loops. Simply quantizing and compacting produces code that under-utilizes resources at the beginning and end of every quantized iteration.

If a loop could be fully unrolled and compacted (making the quantization box the entire iteration space of that loop) then the start-up and wind-down costs would be incurred only at the beginning and end of the loop's execution. Full unrolling is generally undesirable or impossible; however, there is a transformation, *perfect pipelining*, which achieves the effect of full unrolling and compaction of a loop[AN88b,Aik88]. Perfect Pipelining combines very fine-grain parallelism with the pipelining of loop iterations. The idea behind perfect pipelining is that a loop's dependencies encode some repeating behavior, and unrolling and compacting the loop is guaranteed to exhibit that repeating behavior with a finite—and in practice small—amount of unrolling. Informally, this repeating behavior is called the *pattern* of the loop. Once discovered, the loop body can be replaced by this pattern, and, given enough resources, the loop obtained runs in optimal time, subject to the data-dependencies and the given transformations. Further unrolling and compaction of the loop cannot yield better speedups. Thus, despite the limited unrolling, the running time of the transformed loop is identical to what might be obtained by full unrolling of the loop and full fine-grain parallelization, if full unrolling were feasible. The technique is very general; perfect pipelining applies to loops with arbitrary flow of control and with arbitrary resources.[5]

A full description of perfect pipelining and proofs of correctness may be found in [Aik88]; we present an example to illustrate the technique. Refer again to Figure 7. Note that the

[5]Perfect pipelining does not itself solve the difficult problem of scheduling with constrained resources; however, perfect pipelining can be applied in conjunction with a heuristic scheduler.

```
for i ← 0 to N_i by 3 do
    begin
        [i, 1]
        [i, 2]      [i + 1, 1]
        for j ← 0 to N_j − 2 by 3
            begin
                [i, j + 2]      [i + 1, j + 1]      [i + 2, j]
            end;
        [i + 1, N_j]      [i + 2, N_j − 1]
        [i + 2, N_j]
    end;
```

Figure 9: Loop after perfect pipelining is applied to the inner loop.

third and fourth line of the compacted loop are the same except for increments to the index variable i. Figure 8 shows that further unrolling and compaction of the inner loop produces more parallel statements of the same type. Perfect Pipelining infers this pattern from the loop in Figure 7; the resulting code is shown in Figure 9. The code outside of the inner loop forms a prologue and epilogue to the fully pipelined computation of the inner loop. Note that the epilogue/prologue code is an immediate byproduct of detecting the pattern—in Figure 9, the prologue is everything before the pattern, the epilogue is everything after the pattern.

This loop is much better than the loop in Figure 7 and no more expensive to compute. However, it is still not as good as the wavefront method—the start-up and wind-down penalty is still paid for every iteration of the outer loop. Applying perfect pipelining to the outer loop eliminates this inefficiency. This is not quantization. The outer loop is being treated as a single loop; the inner loop is treated as a single statement which cannot be scheduled in parallel with any other statement (it cannot anyway, due to resource constraints). The pattern overlaps the wind-down phase of one iteration of the j loop with the start-up phase of the next. The same procedure can be applied to any additional outer loops. The final code is shown in Figure 10. Figure 11 shows how this loop relates to the wavefront method. Each inner loop iteration executes "small" wavefronts. The end of each inner loop iteration is overlapped with the beginning of the next.[6]

[6]The perfect dovetailing of the wind-down code for one iteration and the start-up code for the next is not accidental; this is a result of the regularity of the iteration space and always occurs.

```
[1,1]
[2,1]    [1,2]
for i ← 0 to N_i − 3 by 3 do
    begin
        for j ← 0 to N_j − 2 by 3 do
            begin
                [i,j+2]      [i+1,j+1]      [i+2,j]
            end;
        [i+3,1]      [i+1,N_j]      [i+2,N_j−1]
        [i+3,2]      [i+4,1]        [i+2,N_j]
    end;
for j ← 0 to N_j − 2 by 3 do
    begin
        [N_i−2,j+2]      [N_i−1,j+1]      [N_i,j]
    end;
[N_i−1,N_j]    [N_i,N_j−1]
[N_i,N_j]
```

Figure 10: Loop after perfect pipelining is applied to the outer loop.

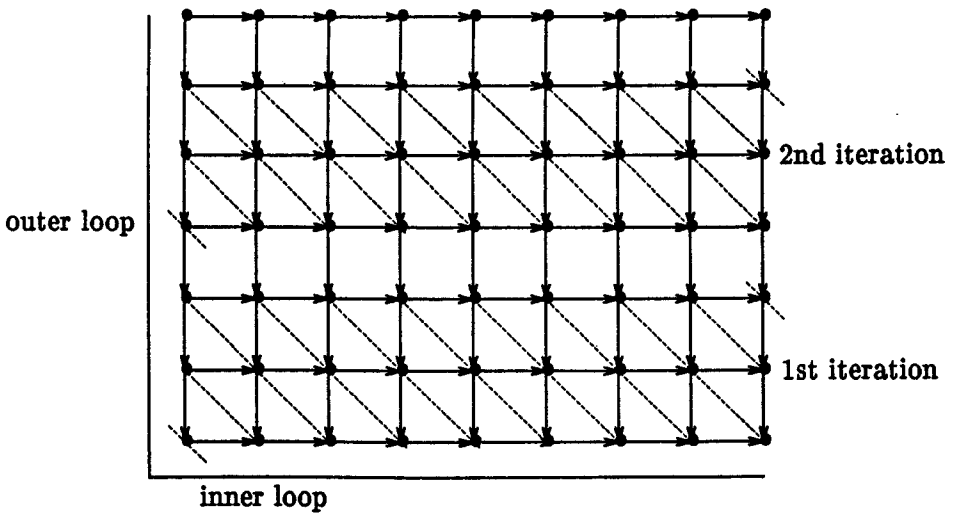


Figure 11: Execution of the final loop.

```
(* Given nested loops where mitred quantization is legal *)
Compute mitred quantization;
repeat
    increase unrolling of the outer loop;
    repeat
        increase unrolling of the inner loop;
        compact
    until pattern detected;
    if pattern utilizes resources sufficiently
        then break  (* leave repeat loop *)
until false;
Apply perfect pipelining to other loops;
Delete empty nodes;
```

Figure 12: A compaction algorithm for nested loops.

We have glossed over many details in this example. In general a non-trivial mitred quantization is required; this presents no additional problems. The start-up and wind-down code that appears after each perfect pipelining transformation is just the code that is left over after the pattern is extracted. We have implicitly assumed that the loop limits of the final loop are divisible by three and that the iteration space is at least three by three. Methods that introduce a small amount of extra code to correct for these assumptions are well known[CCK87].

6 The Algorithm

The algorithm is given in Figure 12. The inner loop is repeatedly unrolled and compacted until a pattern is detected. If the resulting pattern does not fully utilize the machine's resources, then the outer loop is unrolled once and the process repeated. The effect of this procedure is to generate larger and larger pieces of the wavefronts executed by the wavefront method. Because mitred quantization applies under exactly the same conditions as the wavefront method (Lemma 4.1), and because wavefronts can be arbitrarily large (limited only by the size of the iteration space), a large enough quantization combined with perfect pipelining fills the machine's resources and the algorithm terminates. Pipelining on the other loops sustains utilization throughout execution of the nested loops.

This technique extends naturally to higher dimensions in the same manner as the wave-

front method. In practice, however, it is usually unnecessary to use more than two dimensions. As observed above, if the problem size is large relative to the size of the machine, then sufficient parallelism exists in two dimensions for full utilization.

7 Conclusion

We have shown that the effect of the wavefront method can be achieved as a compaction transformation. Besides the theoretical interest, this has practical implications. It allows the power of the wavefront method to be used in compaction-based compilers. Although we have considered only single statement recurrences in this paper, perfect pipelining and compaction extend immediately to general loop bodies. In fact, on more complex loops this algorithm should perform better than the wavefront method, because the underlying fine-grain compaction exploits parallelism between individual statements both inside and across iterations. Thus even in cases where the wavefront method would not apply, this method may find substantial parallelism.

References

[Aik88] A. Aiken. *Compaction-Based Parallelization.* PhD thesis, Cornell, 1988. Department of Computer Science Technical Report No. 88-922.

[AK82] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. In Kai Hwang, editor, *Supercomputers: Design and Applications*, pages 188–203, IEEE Computer Society Press, Silver Spring, MD, 1982.

[AN87] A. Aiken and A. Nicolau. *Loop Quantization: An Analysis and Algorithm.* Technical Report 87-821, Cornell University, March 1987.

[AN88a] A. Aiken and A. Nicolau. A development environment for horizontal microcode. *IEEE Transactions on Software Engineering*, 14(5):584–594, May 1988.

[AN88b] A. Aiken and A. Nicolau. Perfect Pipelining: a new loop parallelization technique. In *Proceedings of the 1988 European Symposium on Programming*,

pages 221–235, Springer Verlag Lecture Notes in Computer Science no. 300, March 1988. Also available as Cornell Technical Report TR 87-873.

[CCK87] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 295–304, August 1987.

[Cyd87] *Technical Summary.* Cydrome Inc., Palo Alto, Ca., 1987.

[Cyt86] R. Cytron. Doacross: beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844, August 1986.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, pages 37–47, June 1984.

[Fis79] J. A. Fisher. *The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources.* PhD thesis, New York University, 1979.

[KKP*81] D. J. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 1981 SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[Kuc76] D. J. Kuck. Parallel processing of ordinary programs. In *Advances in Computers*, pages 119–179, Academic Press, New York, 1976.

[Kuh80] R. H. Kuhn. *Optimization and Interconnection Complexity for: Parallel Processors, Single-Stage Networks, and Decision Trees.* PhD thesis, University of Illinois at Urbana-Champaign, 1980. Dept. of Computer Science Rpt. 80-1009.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[Mur71] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, University of Illinois at Urbana-Champaign, 1971.

[Nic85] A. Nicolau. Uniform parallelism exploitation in ordinary programs. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 614–618, August 1985.

[Nic87] A. Nicolau. Loop Quantization, or unwinding done right. In *Proceedings of the 1987 ACM International Conference on Supercomputing*, Springer Verlag Lecture Notes in Computer Science no. 298, May 1987.

[PW86] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, October 1982.

[Wol87] M. Wolfe. *Loop Skewing: the Wavefront Method Revisited.* Technical Report, University of Illinois at Urbana-Champaign, April 1987.